KÜNSTLICHE NEURONALE NETZWERKE

Joern Ploennigs · AI4SC

Künstliche Neuronale Netzwerke haben sich in den letzten Jahren zu den wichtigsten Verfahren im Bereich des maschinellen Lernens und der künstlichen Intelligenz entwickelt. Dabei gehören zu den ältesten und grundlegendsten Algorithmen im Bereich des maschinellen Lernens und der Mustererkennung. Die ersten Grundlagen wurden bereits in den 1940er Jahren entwickelt, als ein Neurophysiologe zusammen mit einem Mathematiker ein Modell für das menschliche Denken vorschlug [McCulloch & Pitts, 1943], das später u.a. von Rosenblatt zum Perzeptron weiterentwickelt wurde [Rosenblatt, 1958].

Sie sind inspiriert von den biologischen neuronalen Netzwerken des menschlichen Gehirns und versuchen dieses mathematisch nachzuempfinden und somit auch ähnliche Lernerfolge zu erzielen. Sie bestehen aus vielen miteinander verbundenen "Neuronen", die in Schichten organisiert sind und zur Verarbeitung und Analyse komplexer Datenmuster verwendet werden.

Sie werden heutzutage in vielen Anwendungsfällen genutzt insbesondere bei der Verarbeitung unstrukturierter Daten wie Bilder, Texte oder Sprache. Im Bauwesen werden sie eingesetzt zur automatisierten Rissanalyse in Betonbauwerken mittels Bildverarbeitung, zur Suche nach anomalen Vibrationen oder Dehnungsgeräuschen durch Schallemissionsverfahren oder zur Prognose von Feuchte-/Temperaturverläufen in Bauteilen über neuronale Zeitreihenmodelle.

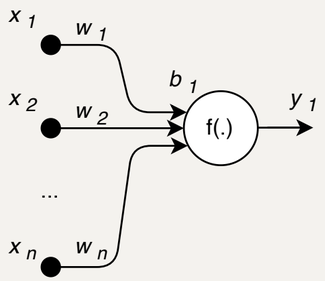
Evaluation

Füllt bitte die Evaluation aus und helft die Vorlesung zu verbessern:

GRUNDLAGEN

Neuronen

Neuronen sind die grundlegenden Bausteine eines neuronalen Netzwerks. Ein Neuron empfängt Eingaben, verarbeitet sie und gibt eine Ausgabe weiter. Die Verarbeitung erfolgt durch eine Aktivierungsfunktion $f(\cdot)$, die den linearen Eingang in eine nichtlineare Ausgabe umwandelt. Das einfachste Neuron ist ein Perceptron [Rosenblatt, 1958]. Seine Grundstruktur besteht aus mehreren Komponenten, die in ähnlicher Form auch in modernen Neuronen vorkommen:



- **Eingaben (Inputs, $x_1, x_2, \ldots, x_n$):** Signale oder Datenpunkte, die in das Neuron eingespeist werden.
- **Gewichte (Weights, $w_1, w_2, \ldots, w_n$):** Faktoren, die die Bedeutung jedes Eingangs steuern.
- **Bias (Bias, b):** Ein zusätzlicher Parameter, der den Schwellenwert für die Aktivierung anpasst.
- **Aktivierungsfunktion (Activation Function, $f(\cdot)$):** Eine Funktion, die den Summenwert der gewichteten Eingaben und des Bias in eine Ausgabe umwandelt.

Die Ausgabe eines Neurons wird durch die folgende Formel berechnet:

$$y = f\left(b + \sum_{i=1}^{n} w_i x_i\right)$$

Damit gleicht das Neuron in dieser Formel einem *Generalisierten Linearem Modell (GLM)*. Dies trifft aber nur für das einfachste neuronale Netzwerk, das Perceptron, zu. Modernere KNN bestehen aus komplexeren Modellen, die komplexe nichtlineare Zusammenhänge modellieren können. Die Aktivierungsfunktion spielt dabei eine entscheidende Rolle, da sie die Nichtlinearität in das Modell einführt und es dem Netzwerk ermöglicht, komplexe Muster in den Daten zu lernen.

Bei GLM zielt die Transformationsfunktion $f(\cdot)$ darauf ab, die Verteilung der Ausgangsvariable zu verändern. Die meisten Aktivierungsfunktionen bei KNN zielen darauf ab, das Ausgangssignal entweder klarer zu differenzieren (aktiv/inaktiv) oder nichtlinear zu transformieren, um komplexe Zusammenhänge zu lernen. Einige gängige Aktivierungsfunktionen sind:

- **Sigmoid:** Diese Funktion kennen wir von der logistischen Regression und GLM. Sie transformiert den Eingang in einen Wert zwischen 0 und 1, was sie besonders geeignet für die Ausgabe von Wahrscheinlichkeiten macht.

$$f(x) = (1 + e^{-x})^{-1}.$$

- **Tanh (Hyperbolic Tangent):** Diese Funktion transformiert den Eingang in einen Wert zwischen -1 und 1 und wurde historisch in früheren neuronalen Netzen verwendet. Sie kann in Aufgaben nützlich sein, bei denen sowohl positive als auch negative Werte relevant sind.

$$f(x) = \tanh(x).$$

- **ReLU (Rectified Linear Unit):** Sie ist eine einfache und häufig verwendete Funktion, die den Eingang direkt zurückgibt, wenn er positiv ist, und ansonsten Null. ReLU ist bekannt dafür, neuronale Netze schneller und effizienter zu trainieren.

$$f(x) = \max(0, x).$$

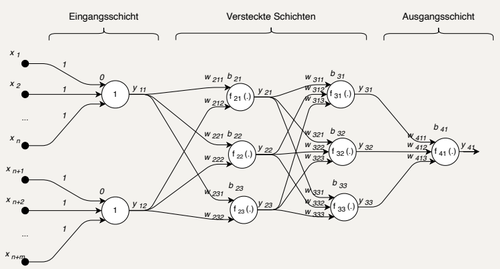
- **Softmax:** Die Softmax-Funktion normalisiert die Ausgabe eines Neurons auf eine Wahrscheinlichkeitsverteilung, wobei die Summe aller Ausgänge 1 ergibt. Sie wird häufig in mehrstufigen Klassifizierungsaufgaben eingesetzt, wo sie die Wahrscheinlichkeiten aller Klassen gleichzeitig ausgibt.

$$f(x_i) = e^{x_i}\left(\sum_{j=1}^{K} e^{x_j}\right)^{-1}.$$

Schichten

Die Fähigkeit komplexe Zusammenhänge zu erlernen, ergibt sich vor allem daraus, dass Neuronale Netzwerke aus mehreren Schichten von Neuronen bestehen, die in einer bestimmten *Architektur* angeordnet sind.

- *Eingabeschicht* (*Input Layer*) nimmt die Rohdaten entgegen und transformiert sie für die nachfolgenden Schichten .
- *Verborgene Schichten* (*Hidden Layers*) enthalten Neuronen, die die eingehenden Daten verarbeiten und transformieren. Bei modernen *Tiefen Neuronalen Netzwerken* (*DNN - Deep Neural Networks*) ist die Anzahl der Schichten hoch (3-100 Schichten).
- *Ausgabeschicht* (*Output Layer*) transformieren die Ausgaben des Netzwerk in die Ergebnisse. Die Anzahl der Neuronen in der Ausgabeschicht hängt von der Art des zu lösenden Problems ab.



Trainingsprozess

Der Grund, weshalb Neuronale Netzwerke fast 60 Jahre lang nicht genutzt wurden, lag in Skalierbarkeit des Trainings. Das oben abgebildete Beispiel zeigt bereits, die hohe Anzahl an 25 Parametern (Gewichte und Bias) für ein kleines Netzwerk mit vier Schichten. Leistungsfähige Netze haben bis zu Milliarden an Parametern.

Aufgrund der Komplexität können die Parameter nicht direkt berechnet werden und werden deshalb normalerweise iterativ angenähert. Dieser Prozess umfasst typischerweise folgende Schritte:

01 **Zufällige Initialisierung** aller Parameter w und b mit Zufallszahlen.

02 **Vorwärtsausbreitung (Forward Propagation):** Die Eingabedaten werden durch das Netzwerk geleitet und die Ausgabe geschätzt. Jedes Neuron berechnet seine Aktivierungsfunktion basierend auf den gewichteten Eingaben und dem Bias mit Hilfe der oben angegeben Formel.

03 **Berechnung des Fehlers (Loss Calculation):** Der Fehler oder Verlust wird durch eine Verlustfunktion berechnet, die die Differenz zwischen der vorhergesagten Ausgabe und der tatsächlichen Ausgabe misst. Die passende Verlustfunktionen richtet sich dabei nach dem Datentyp der Ausgangsvariable. Typische Verlustfunktionen kennen wir bereits:

   04 *Mean Square Error* für numerische Daten (siehe Lineare Regression).

$$E^{\mathrm{MSE}} = \frac{1}{n}\sum_{j=1}^{n}(\hat{y}_j - y_j)^2$$

   05 *Kreuzentropie (LogLoss)* für binäre oder kategorische Daten (siehe Logistische Regression).

$$E^{\mathrm{LL}} = -\frac{1}{n}\sum_{j=1}^{n}\left[y_j \log(\hat{y}_j) + (1-y_j)\log(1-\hat{y}_j)\right]$$

01 **Rückwärtsausbreitung (Backpropagation):** Der Fehler wird durch das Netzwerk rückwärts propagiert, um die Gradienten der Verlustfunktion bezüglich der Gewichte und Biases zu berechnen. Dabei wird im Wesentlichen der Fehler rückwärts im Netzwerk verteilt, so dass Neuronen, die einen großen Anteil an der Vorhersage haben (also einen hohen Eingangswert und eine steile Aktivierungsfunktion an dieser Stelle), ein größerer Anteil am Fehler zugeteilt wird. Der Gradient $\nabla E(w_{ij})$ des Gewichtes w_{ij} kann mit der Kettenregel berechnet werden:

$$\nabla E(w_{ij}) = \frac{\partial E}{\partial w_{ij}}$$

$$\nabla E(b_i) = \frac{\partial E}{\partial b_i}$$

Die entsprechende partielle Ableitung richtet sich prinzipiell nach der Aktivierungsfunktion und wird in modernen Bibliotheken meist analytisch mit der Kettenregel sowie automatischer Differentiation berechnet.

02 **Gewichtsaktualisierung (Weight Update):** Die Gewichte und Biases werden mithilfe eines Optimierungsalgorithmus wie Gradient Descent oder Adam angepasst, um den Fehler zu minimieren. Beim Gradient Descent geschieht dies durch Anwendung der berechneten Gradienten auf die Gewichte und Biases mit Hilfe der Lernrate α:

$$w_{ij} = w_{ij} - \alpha * \nabla E(w_{ij})$$

$$b_i = b_i - \alpha * \nabla E(b_i)$$

03 **Abbruchkriterium**: Wiederholung der Schritte 2.-5. bis ein Konvergenzkriteriums erfüllt wurde oder eine maximale Anzahl an Iterationen erreicht wird.

Epochen

Der Trainingsprozess wird in der Regel über mehrere **Epochen** wiederholt, wobei jede Epoche eine Durchführung des gesamten Trainingsdatensatzes darstellt. Dies ermöglicht es dem Netzwerk, die Gewichte iterativ zu verbessern und seine Leistung auf neuen Daten zu generalisieren. Ein entscheidender Grund für den heutigen Erfolg von Neuronalen Netzwerken ist, dass diese Berechnung der Gewichte hoch parallelisierbar ist und somit sehr gut auf modernen Graphikkarten parallelisiert werden können.

Overfitting

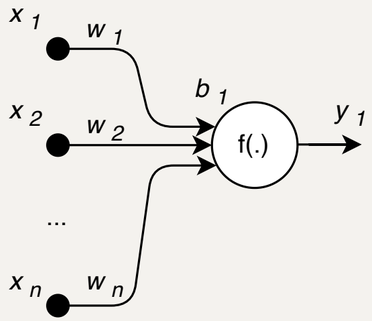
Ein häufiges Problem beim Training neuronaler Netzwerke ist Overfitting, bei dem das Netzwerk die Trainingsdaten zu gut lernt und auf neuen Daten schlecht generalisiert. Um Overfitting zu vermeiden und die Leistung des Netzwerks zu verbessern, werden verschiedene Techniken der Regularisierung eingesetzt:

- **Regularisierung (z.B. L1, L2):** Hinzufügen eines Regularisierungsterms zur Verlustfunktion, um die Komplexität des Modells zu kontrollieren.
- **Dropout:** Zufälliges Ausschalten von Neuronen während des Trainings, um die Abhängigkeit von bestimmten Neuronen zu reduzieren.
- **Datenaugmentation:** Erweiterung des Trainingsdatensatzes durch zufällige Transformationen der Daten.

ARCHITEKTUREN

Vorwärtsgerichtete Netze

Die einfachsten Formen neuronaler Netzwerke sind *Vorwärtsgerichtete Neuronale Netze* (*FNN - Feedforward Neural Networks*). Bei ihnen fließen die Daten in eine Richtung von der Eingabeschicht zur Ausgabeschicht, ohne rückwärts gerichtete Datenflüsse. Sie entsprechen den oben dargestellten Netzwerken. Sie sind besonders geeignet bei einfachen Problemen, wo keine Autokorrelation oder komplexe Muster vorherrschen.

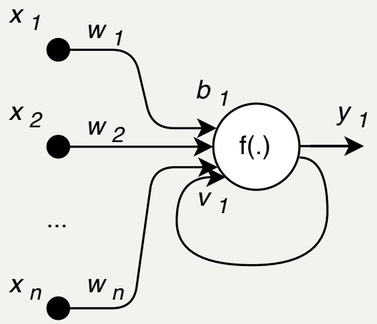


Rekurrente Netze

Rekurrente oder *rückgekoppelte Neuronale Netze* (*RNN - Recurrent Neural Networks*) sind neuronalen Netzen, die sich von Feedforward-Netzen dadurch unterscheiden, dass sie Verbindungen zwischen Neuronen einer Schicht zu Neuronen derselben oder vorheriger Schichten ermöglichen. Diese Art der Verschaltung ähnelt der bevorzugten Struktur neuronaler Netze im Gehirn, insbesondere im Neocortex. RNNs sind ideal für sequenzielle Daten wie Zeitreihen oder Text, da sie über Schleifen verfügen, die Informationen über Zeitpunkte (oder Wortfolgen/Sätze) hinweg speichern. Dadurch lässt sich z.B. gut die Autoregression in Zeitreihen oder komplette Sätze erlernen.

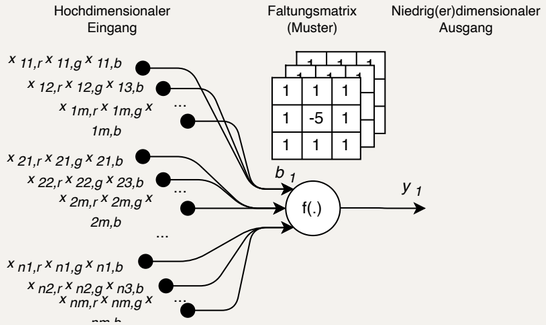
$$h_t = f\left(b + \sum_{i=1}^{n} w_i x_{i,t} + v\, h_{t-1}\right)$$

Hierbei ist h_t der versteckte Zustand (*hidden state*) zum aktuellen Zeitpunkt t und h_{t-1} der versteckte Zustand des vorherigen Zeitpunkts, der über das Gewicht v zurückgeführt wird.



Faltungsnetze

Faltungsnetze (*CNN - Convolutional Neural Networks*) sind besonders effektiv bei der Verarbeitung von höherdimensionalen Daten wie Bilddaten oder geospatialer Daten. Sie verwenden Faltungsoperationen, um Merkmale (Kanten, Reliefs, Muster, etc.) aus Eingabebildern zu extrahieren oder zu erlernen und gleichzeitig einen höherendimensionalen Eingang auf einen niedrigdimensionaleren Ausgang zu reduzieren, der diese Merkmale encodiert. Dadurch repräsentiert der Ausgang nur die Relevanz des extrahierten oder gelernten Merkmals und es muss nicht das Gesamtbild gelernt werden. Durch das Schichten mehrerer versteckter Faltungsschichten hintereinander werden so zuerst Merkmale extrahiert und dann Muster daraus im Bild gelernt (z.B. die Form eines Hundes). Auf die Faltung und die Bildverarbeitung werden wir im Nachfolgekurs "Künstliche Intelligenz und Foundation Modelle" noch vertieft eingehen.



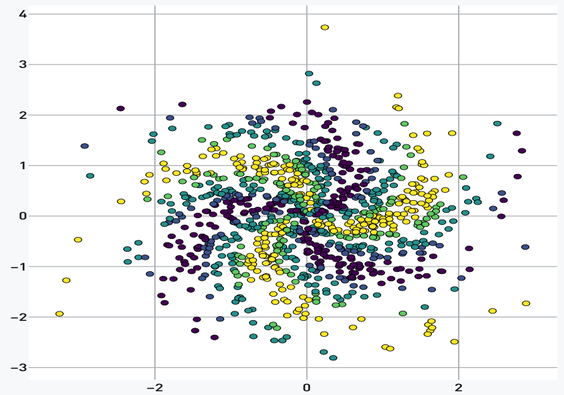
NEURONALE NETZWERKE IN PYTHON

Neuronale Netzwerke in `sklearn`

Wir demonstrieren neuronale Netzwerke anhand einer erweiterten Version des Yin-Yang-Beispiels aus der Klassifikation indem wir die Anzahl der Farben auf 4 Kategorien und die Verwirbelung erhöhen. Wieder besteht die Aufgabe darin, die richtige Farbe basierend auf dem Punkt vorherzusagen, also eine Klassifikationsaufgabe. Wir erstellen die Daten wie folgt:

```
import numpy as np # Import von NumPy
import pandas as pd # Import von Pandas
import plotly.express as px # Import von Plotly
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

np.random.seed(33)
N = 1000  # Anzahl der Punkte
X1 = np.random.normal(size=N)
X2 = np.random.normal(size=N)
X = np.column_stack((X1, X2))

alpha = np.arctan2(X2, X1)
r = np.sqrt(X1**2 + X2**2)

## Teile the sum of a sin and cosine into 5 intervals
category = pd.cut(np.sin(3*alpha + 2*r) + np.cos(3*alpha + 2*r),
           bins=[-1.5, -1.1, -0.6, 0.6, 1.1, 1.5],
           labels=[0, 1, 2, 3, 4])
y = category.astype(int)

# Teile in Test und Trainings-Datensatz
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
# Darstellung des Datensatzes
fig=px.scatter(x=X2, y=X1, color=y, width=600, height=600)
fig.update_traces(marker_line=dict(width=.3, color="black"))
fig.show()
```



Das Bild zeigt ein komplexes Muster von Spiralarmen in fünf verschiedenen Farben. Dreiarmige gelbe und blauen Arme entsprechen den größten und kleinsten Werten, während sechsarmige Arme in verschiedenen Rot- und Violetttönen den dazwischen liegenden Werten entsprechen. Diese Entscheidungsgrenzen sind sehr schwer mit einfachen Modellen wie logistischer Regression, SVM oder sogar Bäumen zu erfassen, da die Spiralarme stark nicht-konvexe und ineinander verschlungene Klassenregionen bilden: Bäume können solche Regionen nur grob durch viele achsenparallele Splits annähern, und ein SVM mit einem einzelnen festen Kernel findet keine Transformation, die alle Spiralarme gleichzeitig sauber trennt. Prüfen wir das noch einmal mit einer kleinen Visualisierungsfunktion, die wir auch in der Klassifikation genutzt haben.

Random Forest

Testen wir einen Random Forest, so erhalten wir zwar ein gutes Modell, dass allerdings overfittet und schlecht generalisiert.

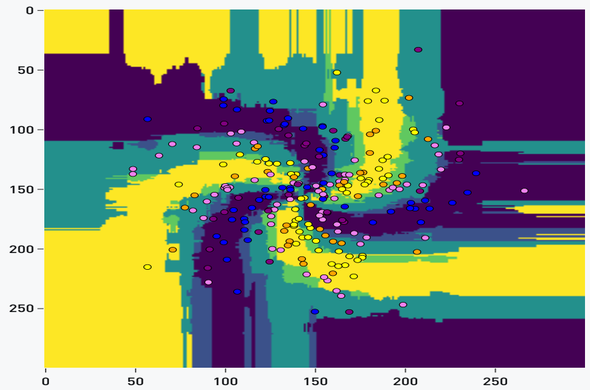
```
from sklearn.ensemble import RandomForestClassifier

m = RandomForestClassifier()
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.772
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[52,  0,  2,  0,  0],
       [10, 19,  5,  0,  0],
       [ 4,  7, 50,  9,  0],
       [ 0,  0, 10, 24,  4],
       [ 0,  0,  3,  3, 48]])
```

```
DBPlot(m, X_test, y_test)
```



SVM

Ein SVM mit radialem Kernel erkennt die Struktur besser, aber hat einen deutlich höheren Fehler.

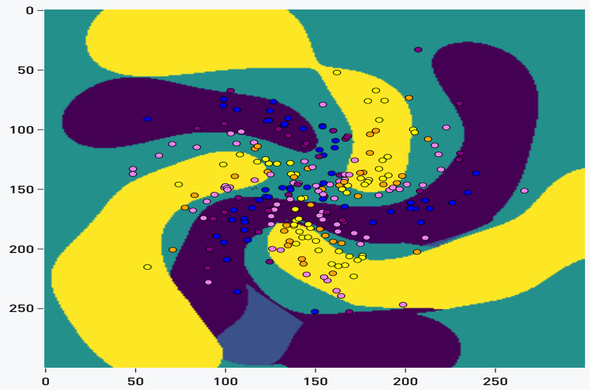
```
from sklearn.svm import SVC

m = SVC(kernel="rbf", gamma=1)
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.544
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[43,  0, 11,  0,  0],
       [19,  0, 15,  0,  0],
       [11,  0, 52,  0,  7],
       [ 1,  0, 11,  0, 26],
       [ 3,  0, 10,  0, 41]])
```

```
DBPlot(m, X_test, y_test)
```



Multi-Layer Perceptrons in `sklearn`

Allerdings können neuronale Netzwerke (und KNN) deutlich besser damit umgehen.

sklearn implementiert einfache Feed-Forward-Neuronale Netzwerke, so genannte *Multi-Layer Perceptrons*. Dies sind einfache dichte Feed-Forward-Netzwerke mit einer beliebigen Anzahl von versteckten Schichten. Auch wenn sie strukturell einfache neuronaler Netzwerke sind, sind sie dennoch leistungsfähig genug für viele Aufgaben.

Wie bei anderen fortgeschrittenen Modellen bietet *sklearn* zwei unterschiedlichen Klassen für Klassifikationsaufgaben `MLPClassifier` und für Regressionsaufgaben `MLPRegressor`. Die grundlegende Verwendung dieser Modelle ist identisch zu den anderen *sklearn*-Modellen.

Die wichtigsten Argumente für den `MLPClassifier` sind:

```
MLPClassifier(hidden_layer_sizes, activation, max_iter, alpha)
```

Von diesen ist **hidden_layer_sizes** der zentralste. Dies beschreibt die Anzahl der versteckten Schichten (die Größe der Eingabe- und Ausgabeschicht wird automatisch aus den Daten bestimmt). Es handelt sich um ein Tupel, das die Anzahl der Knoten für jede versteckte Schicht angibt, daher gibt die Länge des Tupels auch die Anzahl der versteckten Schichten an. Zum Beispiel bedeutet `hidden_layer_sizes = (32, 16)` zwei versteckte Schichten, die erste mit 32 und die folgende mit 16 Knoten. **activation** beschreibt die Aktivierungsfunktion. Es empfiehlt sich meist es bei der Standardfunktion `relu` zu lassen, außer es gibt die oben beschriebenen Gründe für andere Aktivierungsfunktionen. **alpha** ist der L2-Regularisierungsparameter und **max_iter** gibt die maximale Anzahl von Iterationen an, bevor die Optimierung stoppt. In moderneren Bibliotheken wie Keras oder PyTorch wird die L2-Regularisierung meist nicht global, sondern explizit pro Schicht angegeben, z.B. mit `kernel_regularizer=regularizers.L2(alpha)` in Keras.

Einfaches Neuronales Netzwerk

Lassen Sie uns nun die Verwendung von `MLPClassifier` an den Yin-Yang Beispiel demonstrieren. Beginnen wir mit einem einfachen Perzeptron mit einer einzigen versteckten Schicht von 20 Knoten. Daher verwenden wir `hidden_layer_sizes=(20,)`, ein Tupel mit nur einer Zahl. Das Anpassen des Modells und die Vorhersage sind größtenteils ähnlich wie bei den anderen *sklearn*-Funktionen, daher diskutieren wir dies hier nicht. Wir erhöhen auch die Anzahl der Iterationen, da die Standardanzahl von 200 in diesem Fall zu gering ist:

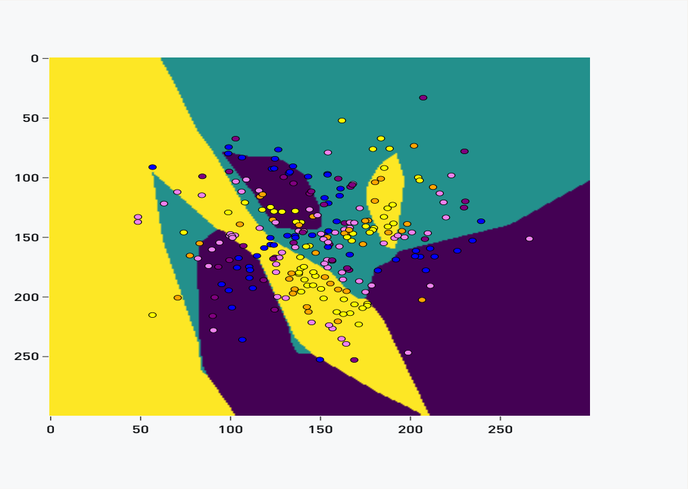
```
from sklearn.neural_network import MLPClassifier

m = MLPClassifier(hidden_layer_sizes = (20,), max_iter=10000)
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.368
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[32,  0, 14,  0,  8],
       [13,  0, 13,  0,  8],
       [14,  0, 29,  0, 27],
       [ 2,  0, 19,  0, 17],
       [ 1,  0, 22,  0, 31]])
```

```
DBPlot(m, X_test, y_test)
```



Dieses einfache Modell hat zuerst auch nur eine niedrige Genauigkeit. Das liegt daran, dass es zu einfach ist, nur eine einzelne kleine versteckte Schicht reicht nicht aus, um das komplexe Spiralmuster gut zu modellieren. Was sich gut in der Visualisierung erkennen lässt. Wir können dort sehen, dass das Modell die Idee korrekt einfängt: Spiralen in verschiedenen Farben, aber die Form der Spiralen ist nicht genau genug.

Tiefes Neuronales Netzwerk

Lassen Sie uns das Modell mit einem leistungsstärkeren Netzwerk wiederholen:

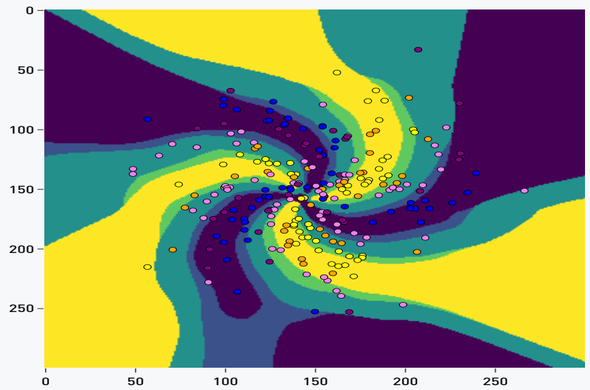
```
m = MLPClassifier(hidden_layer_sizes = (256, 128, 64), max_iter=10000)
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.908
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[53,  1,  0,  0,  0],
       [ 6, 22,  6,  0,  0],
       [ 1,  0, 66,  3,  0],
       [ 0,  0,  4, 33,  1],
       [ 0,  0,  0,  1, 53]])
```

```
DBPlot(m, X_test, y_test)
```



Jetzt sind die Ergebnisse sehr gut mit einer Genauigkeit von ca. 95 %. Die visuelle Inspektion bestätigt, dass das leistungsstärkere neuronale Netzwerk die Gesamtstruktur des Modells gut erfassen kann.

Die angepassten Netzwerkmodelle haben verschiedene Methoden und Attribute, z.B. gibt `coefs_` die Modellgewichte aus (als Liste von Gewichtsmatrizen, eine für jede Schicht), und `intercepts_` gibt die Modell-Biaswerte aus (als Liste von Bias-Vektoren, einer für jede Schicht). Zum Beispiel enthält das oben angepasste Modell den akkumulierten Bias-Vektor für die erste versteckte Schicht:

```
np.sum([b.size for b in m.intercepts_])
```

```
np.int64(453)
```

Während *sklearn* einen einfachen Zugang zu neuronalen Netzwerkmodellen bietet, sind diese Modelle erheblich eingeschränkt. *sklearn* ist damit gut für kleinere Aufgaben und zum schnellen Prototyping geeignet, aber für tiefere Netzwerke, größere Datenmengen und GPU-Beschleunigung braucht man spezialisierte Frameworks wie *tensorflow* oder *pytorch*.

NEURONALE NETZWERKE IN KERAS

Tensorflow and Keras

Tensorflow ist eine Bibliothek, die Berechnungsgraphen implementiert, die automatisch Gradienten berechnen können. Sie ermöglicht auch Berechnungen auf der GPU durchzuführen, was potenziell zu erheblichen Geschwindigkeitsverbesserungen gegenüber CPU-Berechnungen führen kann. In vielerlei Hinsicht ähnelt sie *numpy*, wo man Matrizen und Tensoren erstellen und manipulieren kann. Allerdings ist sie aufgrund des Berechnungsgraphen-Ansatzes und der GPU-bezogenen Überlegungen viel schwieriger zu verwenden.

Keras ist ein TensorFlow-Front-End, ein Submodul, das einen viel benutzerfreundlicheren Zugang zum Aufbau neuronaler Netzwerke bietet. Die Funktionalität umfasst eine Vielzahl von Netzwerkschichten, bei denen man die entsprechenden Parameter anpassen kann. Beim Aufbau des Netzwerks kann man einfach die Schichten hintereinander hinzufügen, die Verbindung zwischen den Schichten wird von Keras selbst übernommen. Eine gute Quelle für die Keras-Dokumentation ist die API-Referenzdokumentation.

Beispielnetzwerk in *keras*

Lassen Sie uns das Farbspiralbeispiel aus dem letzten Abschnitt von *sklearn* neu implementieren, diesmal jedoch mit *keras*. Es wird einige bemerkenswerte Unterschiede geben:

Der Aufbau des Netzwerks selbst ist in Keras anders. Keras berechnet nicht die Größe der Eingabe- und Ausgangsschichten aus den Daten. Beide müssen vom Benutzer angegeben werden. Schließlich sagt Keras nur Wahrscheinlichkeiten für alle Kategorien voraus, daher müssen wir Code hinzufügen, der die Spalte (Kategorie) mit der höchsten Wahrscheinlichkeit findet.

Das Modell erzeugen

Zuerst der wichtigste Schritt: **Aufbau** und **Kompilierung** des Modells. Dies unterscheidet sich sehr von der Vorgehensweise in sklearn. Lassen Sie uns ein sequentielles Modell mit dichten Schichten aufbauen, die gleiche Architektur, die wir mit *sklearn* erstellt haben:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

# sequential (not recursive) model (one input, one output)
model=Sequential()
model.add(Input(shape=(2,)))
model.add(Dense(512, activation="relu"))
model.add(Dense(256, activation="relu"))
model.add(Dense(64, activation="relu"))
nCategories = len(np.unique(category))
model.add(Dense(nCategories, activation="softmax"))
```

Das richtige Festlegen der Eingabeformen und Ausgabeknoten ist eine der Hauptursachen für Frustration, wenn man anfängt, mit *keras* zu arbeiten. Die Fehlermeldungen sind lang und nicht besonders hilfreich, und es ist schwer zu verstehen, was das Problem ist. Hier ist eine Checkliste, die durchgearbeitet werden kann, wenn etwas nicht funktioniert:

- Ist die Eingabeform explizit definiert, z.B. mit `Input(shape=...)` am Anfang des Modells?
- Stellt diese Form korrekt eine einzelne Instanz der Eingabedaten dar?
- Haben Sie die richtige Anzahl von Knoten in der Softmax-aktivierten Ausgangsschicht?

Training des Modells

Die nächste Aufgabe besteht darin, das Modell zu kompilieren und anzupassen. Keras-Modelle müssen kompiliert werden - was wir bisher eingerichtet haben, ist nur eine Beschreibung des Modells, nicht das tatsächliche Modell, das für TensorFlow-Tensoren eingerichtet ist und möglicherweise für die GPU-Ausführung. Wir können das Modell wie folgt kompilieren:

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	1,536
dense_1 (Dense)	(None, 256)	131,328
dense_2 (Dense)	(None, 64)	16,448
dense_3 (Dense)	(None, 5)	325

```
Total params: 149,637 (584.52 KB)
Trainable params: 149,637 (584.52 KB)
Non-trainable params: 0 (0.00 B)
```

```
None
```

Die drei wichtigsten Argumente sind

- `loss` beschreibt die Modellverlustfunktion, `sparse_categorical_crossentropy`, im Wesentlichen die Log-Likelihood, ist geeignet für Kategorisierungsaufgaben. Der genaue Typ hängt auch davon ab, wie das Ergebnis genau codiert ist.
- `optimizer` ist der Optimierer, der für den stochastischen Gradientenabstieg verwendet werden soll. `adam` und `rmsprop` sind gute Optionen, aber es gibt auch andere Möglichkeiten.
- `metrics` ist eine Metrik, die während der Optimierung ausgewertet und gedruckt wird und einige Rückmeldungen darüber bietet, wie die Auswertung verläuft.

Die letzte Zeile hier druckt die Modellzusammenfassung aus, eine praktische Übersicht darüber, was wir gemacht haben:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               1536
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_3 (Dense)              (None, 5)                 325
=================================================================
Total params: 149,637
Trainable params: 149,637
Non-trainable params: 0
```

Dieses Netzwerk enthält fast 150.000 Parameter, von denen alle trainierbar sind.

Nach erfolgreicher Kompilierung können wir das Modell anpassen:

```
history = model.fit(X_train, y_train, epochs=200, validation_split=0.2)
```

```
Epoch 1/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 10s 574ms/step - accuracy: 0.1250 - loss: 1.6099
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.2700 - loss: 1.5818 - val_accuracy: 0.2667 - val_loss: 1.5422
Epoch 2/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1875 - loss: 1.6414
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3033 - loss: 1.5440 - val_accuracy: 0.2933 - val_loss: 1.5430
Epoch 3/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3125 - loss: 1.4983
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3117 - loss: 1.5272 - val_accuracy: 0.3133 - val_loss: 1.5338
Epoch 4/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.2188 - loss: 1.5069
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3267 - loss: 1.5103 - val_accuracy: 0.3533 - val_loss: 1.5474
Epoch 5/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3750 - loss: 1.4836
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3600 - loss: 1.4918 - val_accuracy: 0.3333 - val_loss: 1.5281
Epoch 6/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.2812 - loss: 1.4906
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3850 - loss: 1.4745 - val_accuracy: 0.3467 - val_loss: 1.5465
Epoch 7/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.2500 - loss: 1.4625
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3617 - loss: 1.4537 - val_accuracy: 0.3733 - val_loss: 1.5475
Epoch 8/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.3250
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3683 - loss: 1.4387 - val_accuracy: 0.3800 - val_loss: 1.5086
Epoch 9/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3750 - loss: 1.4665
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4083 - loss: 1.4103 - val_accuracy: 0.4067 - val_loss: 1.5079
Epoch 10/200

 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.2663
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4133 - loss: 1.3942 - val_accuracy: 0.4333 - val_loss: 1.4964
Epoch 11/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3438 - loss: 1.4868
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4433 - loss: 1.3653 - val_accuracy: 0.4600 - val_loss: 1.4938
Epoch 12/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5938 - loss: 1.1625
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4417 - loss: 1.3534 - val_accuracy: 0.4800 - val_loss: 1.4377
Epoch 13/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4688 - loss: 1.3309
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4700 - loss: 1.3176 - val_accuracy: 0.4800 - val_loss: 1.4513
Epoch 14/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.4688 - loss: 1.3534
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4817 - loss: 1.2753 - val_accuracy: 0.4867 - val_loss: 1.3921
Epoch 15/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5000 - loss: 1.3485
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5000 - loss: 1.2475 - val_accuracy: 0.5067 - val_loss: 1.4080
Epoch 16/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6250 - loss: 1.2627
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5283 - loss: 1.1999 - val_accuracy: 0.5400 - val_loss: 1.3399
Epoch 17/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4688 - loss: 1.2639
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5333 - loss: 1.1924 - val_accuracy: 0.5000 - val_loss: 1.3616
Epoch 18/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5625 - loss: 1.0647
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5250 - loss: 1.1591 - val_accuracy: 0.5800 - val_loss: 1.2558
Epoch 19/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.5625 - loss: 1.0130
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5700 - loss: 1.0945 - val_accuracy: 0.5867 - val_loss: 1.2220
Epoch 20/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.9688
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5850 - loss: 1.0564 - val_accuracy: 0.6000 - val_loss: 1.1676
Epoch 21/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.4375 - loss: 1.0739
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6017 - loss: 1.0179 - val_accuracy: 0.6000 - val_loss: 1.1492
Epoch 22/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.9709
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6017 - loss: 0.9904 - val_accuracy: 0.6133 - val_loss: 1.1086
Epoch 23/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.9659
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6367 - loss: 0.9465 - val_accuracy: 0.6667 - val_loss: 1.0316
Epoch 24/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7188 - loss: 0.9169
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6533 - loss: 0.8994 - val_accuracy: 0.6533 - val_loss: 0.9970
Epoch 25/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5625 - loss: 0.9786
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6617 - loss: 0.8593 - val_accuracy: 0.6667 - val_loss: 0.9777
Epoch 26/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.7315
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6983 - loss: 0.8319 - val_accuracy: 0.6933 - val_loss: 0.9466
Epoch 27/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7188 - loss: 0.6948
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7017 - loss: 0.8064 - val_accuracy: 0.6600 - val_loss: 0.9471
Epoch 28/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.7804
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6933 - loss: 0.7929 - val_accuracy: 0.6933 - val_loss: 0.8824
Epoch 29/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7812 - loss: 0.7284
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7233 - loss: 0.7566 - val_accuracy: 0.7067 - val_loss: 0.8491
Epoch 30/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7188 - loss: 0.6787
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6883 - loss: 0.7589 - val_accuracy: 0.7533 - val_loss: 0.7873
Epoch 31/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.6229
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7217 - loss: 0.6962 - val_accuracy: 0.7333 - val_loss: 0.8021
Epoch 32/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7188 - loss: 0.6985
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7600 - loss: 0.6697 - val_accuracy: 0.7200 - val_loss: 0.7649
Epoch 33/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8125 - loss: 0.5404
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7433 - loss: 0.6687 - val_accuracy: 0.7133 - val_loss: 0.7384
Epoch 34/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8438 - loss: 0.4334
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7783 - loss: 0.6398 - val_accuracy: 0.7467 - val_loss: 0.7319
Epoch 35/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.5914
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7867 - loss: 0.6133 - val_accuracy: 0.7933 - val_loss: 0.6836
Epoch 36/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7812 - loss: 0.5988
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8050 - loss: 0.5783 - val_accuracy: 0.7733 - val_loss: 0.7040
Epoch 37/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.6618
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7883 - loss: 0.5698 - val_accuracy: 0.7133 - val_loss: 0.7182
Epoch 38/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.5036
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7750 - loss: 0.5760 - val_accuracy: 0.8133 - val_loss: 0.5871
Epoch 39/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.4813
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7867 - loss: 0.5549 - val_accuracy: 0.7733 - val_loss: 0.7080
Epoch 40/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.4089
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7983 - loss: 0.5319 - val_accuracy: 0.7667 - val_loss: 0.6064
Epoch 41/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.5478
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7533 - loss: 0.5567 - val_accuracy: 0.7067 - val_loss: 0.7194
Epoch 42/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.4501
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7850 - loss: 0.5489 - val_accuracy: 0.7333 - val_loss: 0.6794
Epoch 43/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.4115
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8083 - loss: 0.5292 - val_accuracy: 0.8133 - val_loss: 0.5665
Epoch 44/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.5500
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8150 - loss: 0.4736 - val_accuracy: 0.8200 - val_loss: 0.5827
Epoch 45/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.4186
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8183 - loss: 0.4924 - val_accuracy: 0.7667 - val_loss: 0.6239
Epoch 46/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7500 - loss: 0.4963
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7983 - loss: 0.4928 - val_accuracy: 0.7600 - val_loss: 0.5975
Epoch 47/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7812 - loss: 0.5089
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8217 - loss: 0.4456 - val_accuracy: 0.8333 - val_loss: 0.5245
Epoch 48/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.2970
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8633 - loss: 0.4149 - val_accuracy: 0.8133 - val_loss: 0.4993
Epoch 49/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.3801
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8850 - loss: 0.4098 - val_accuracy: 0.8200 - val_loss: 0.4863
Epoch 50/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.2901
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8767 - loss: 0.3867 - val_accuracy: 0.8133 - val_loss: 0.5779
Epoch 51/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8750 - loss: 0.4579
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8550 - loss: 0.4057 - val_accuracy: 0.8133 - val_loss: 0.5400
Epoch 52/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.3314
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8683 - loss: 0.3769 - val_accuracy: 0.8600 - val_loss: 0.4773
Epoch 53/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.2562
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8783 - loss: 0.3724 - val_accuracy: 0.8600 - val_loss: 0.5001
Epoch 54/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.3494
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8717 - loss: 0.3578 - val_accuracy: 0.8267 - val_loss: 0.5163
Epoch 55/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.3917
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8817 - loss: 0.3670 - val_accuracy: 0.8200 - val_loss: 0.5075
Epoch 56/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8125 - loss: 0.4241
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8567 - loss: 0.3815 - val_accuracy: 0.8533 - val_loss: 0.5593
Epoch 57/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.2333
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.3487 - val_accuracy: 0.8733 - val_loss: 0.5011
Epoch 58/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.3197
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8850 - loss: 0.3530 - val_accuracy: 0.8867 - val_loss: 0.4561
Epoch 59/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.2948
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9150 - loss: 0.3112 - val_accuracy: 0.8667 - val_loss: 0.4426
Epoch 60/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.2099
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3189 - val_accuracy: 0.8000 - val_loss: 0.5081
Epoch 61/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.4671
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8883 - loss: 0.3051 - val_accuracy: 0.8333 - val_loss: 0.4504
Epoch 62/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.3849
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9083 - loss: 0.2932 - val_accuracy: 0.8733 - val_loss: 0.4313
Epoch 63/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2946
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9050 - loss: 0.3061 - val_accuracy: 0.8067 - val_loss: 0.5505
Epoch 64/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.1988
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8933 - loss: 0.3157 - val_accuracy: 0.8200 - val_loss: 0.5363
Epoch 65/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.4268
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8883 - loss: 0.3142 - val_accuracy: 0.7933 - val_loss: 0.4582
Epoch 66/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.2858
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8717 - loss: 0.3356 - val_accuracy: 0.8600 - val_loss: 0.4343
Epoch 67/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.2767
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8450 - loss: 0.3673 - val_accuracy: 0.8533 - val_loss: 0.4890
Epoch 68/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.2347
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8800 - loss: 0.3287 - val_accuracy: 0.8267 - val_loss: 0.4500
Epoch 69/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2608
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2853 - val_accuracy: 0.8467 - val_loss: 0.4165
Epoch 70/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.3009
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.2704 - val_accuracy: 0.8800 - val_loss: 0.3792
Epoch 71/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1735
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9133 - loss: 0.2744 - val_accuracy: 0.8067 - val_loss: 0.4698
Epoch 72/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1993
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9033 - loss: 0.2701 - val_accuracy: 0.8467 - val_loss: 0.4193
Epoch 73/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2826
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9050 - loss: 0.2788 - val_accuracy: 0.8333 - val_loss: 0.4803
Epoch 74/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1750
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.2575 - val_accuracy: 0.8333 - val_loss: 0.5452
Epoch 75/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.2751
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.2797 - val_accuracy: 0.8200 - val_loss: 0.4613
Epoch 76/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2984
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2850 - val_accuracy: 0.8533 - val_loss: 0.4502
Epoch 77/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1958
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9017 - loss: 0.2632 - val_accuracy: 0.8400 - val_loss: 0.4127
Epoch 78/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2073
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9217 - loss: 0.2381 - val_accuracy: 0.9067 - val_loss: 0.3660
Epoch 79/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.2170
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9183 - loss: 0.2526 - val_accuracy: 0.8133 - val_loss: 0.4676
Epoch 80/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2991
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9283 - loss: 0.2487 - val_accuracy: 0.8800 - val_loss: 0.3772
Epoch 81/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.1048
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9200 - loss: 0.2357 - val_accuracy: 0.8733 - val_loss: 0.3950
Epoch 82/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2142
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9133 - loss: 0.2482 - val_accuracy: 0.8533 - val_loss: 0.4011
Epoch 83/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2189
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9000 - loss: 0.2417 - val_accuracy: 0.8867 - val_loss: 0.3730
Epoch 84/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1894
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.2210 - val_accuracy: 0.8400 - val_loss: 0.4141
Epoch 85/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.2254
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.2308 - val_accuracy: 0.8000 - val_loss: 0.4976
Epoch 86/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.2559
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.2178 - val_accuracy: 0.8733 - val_loss: 0.3787
Epoch 87/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2195
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.2084 - val_accuracy: 0.8600 - val_loss: 0.4268
Epoch 88/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2635
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.2014 - val_accuracy: 0.8533 - val_loss: 0.4046
Epoch 89/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8750 - loss: 0.2033
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.2011 - val_accuracy: 0.8600 - val_loss: 0.4304
Epoch 90/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1409
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1970 - val_accuracy: 0.8400 - val_loss: 0.4018
Epoch 91/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8750 - loss: 0.2208
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9133 - loss: 0.2405 - val_accuracy: 0.8667 - val_loss: 0.3977
Epoch 92/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.2740
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.2068 - val_accuracy: 0.8867 - val_loss: 0.4093
Epoch 93/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1547
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.2090 - val_accuracy: 0.8667 - val_loss: 0.3645
Epoch 94/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.2454
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1950 - val_accuracy: 0.8267 - val_loss: 0.4111
Epoch 95/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2271
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1937 - val_accuracy: 0.8733 - val_loss: 0.3982
Epoch 96/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2042
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1798 - val_accuracy: 0.8467 - val_loss: 0.3657
Epoch 97/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1632
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1795 - val_accuracy: 0.8200 - val_loss: 0.4359
Epoch 98/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1604
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.1839 - val_accuracy: 0.8600 - val_loss: 0.3926
Epoch 99/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.2070
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9117 - loss: 0.2277 - val_accuracy: 0.8333 - val_loss: 0.4279
Epoch 100/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.1690
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9283 - loss: 0.1996 - val_accuracy: 0.8533 - val_loss: 0.3927
Epoch 101/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.2214
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1959 - val_accuracy: 0.8800 - val_loss: 0.3805
Epoch 102/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.2796
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1800 - val_accuracy: 0.8800 - val_loss: 0.3821
Epoch 103/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1698
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1693 - val_accuracy: 0.8600 - val_loss: 0.3737
Epoch 104/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1135
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1705 - val_accuracy: 0.8533 - val_loss: 0.3673
Epoch 105/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1027
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9483 - loss: 0.1538 - val_accuracy: 0.8667 - val_loss: 0.4317
Epoch 106/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0931
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1585 - val_accuracy: 0.8867 - val_loss: 0.3784
Epoch 107/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1401
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1835 - val_accuracy: 0.8800 - val_loss: 0.4010
Epoch 108/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1466
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.1904 - val_accuracy: 0.8667 - val_loss: 0.3803
Epoch 109/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1261
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1578 - val_accuracy: 0.8933 - val_loss: 0.3698
Epoch 110/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.1039
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1592 - val_accuracy: 0.8867 - val_loss: 0.4070
Epoch 111/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.2036
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1588 - val_accuracy: 0.8600 - val_loss: 0.3921
Epoch 112/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.1412
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1570 - val_accuracy: 0.8533 - val_loss: 0.4142
Epoch 113/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.1049
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1680 - val_accuracy: 0.8467 - val_loss: 0.4941
Epoch 114/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.1348
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9183 - loss: 0.2280 - val_accuracy: 0.8800 - val_loss: 0.4058
Epoch 115/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.2321
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1733 - val_accuracy: 0.8533 - val_loss: 0.4171
Epoch 116/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.1745
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9217 - loss: 0.1943 - val_accuracy: 0.8933 - val_loss: 0.3574
Epoch 117/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.3095
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2058 - val_accuracy: 0.8933 - val_loss: 0.4108
Epoch 118/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1046
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.2155 - val_accuracy: 0.8467 - val_loss: 0.3931
Epoch 119/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0491
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2458 - val_accuracy: 0.8667 - val_loss: 0.4058
Epoch 120/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.2886
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.1809 - val_accuracy: 0.9133 - val_loss: 0.3486
Epoch 121/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2945
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1700 - val_accuracy: 0.8533 - val_loss: 0.3882
Epoch 122/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2376
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1681 - val_accuracy: 0.8667 - val_loss: 0.3850
Epoch 123/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1598
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1521 - val_accuracy: 0.8467 - val_loss: 0.3957
Epoch 124/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1064
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9633 - loss: 0.1408 - val_accuracy: 0.9067 - val_loss: 0.3436
Epoch 125/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0674
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1579 - val_accuracy: 0.8933 - val_loss: 0.3752
Epoch 126/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.1607
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1491 - val_accuracy: 0.8600 - val_loss: 0.4794
Epoch 127/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0988
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1446 - val_accuracy: 0.8800 - val_loss: 0.3687
Epoch 128/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1620
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.1768 - val_accuracy: 0.8800 - val_loss: 0.4369
Epoch 129/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.1414
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1595 - val_accuracy: 0.8467 - val_loss: 0.4130
Epoch 130/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0912
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.1749 - val_accuracy: 0.8467 - val_loss: 0.4725
Epoch 131/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.1651
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.1954 - val_accuracy: 0.8800 - val_loss: 0.3851
Epoch 132/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1590
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1885 - val_accuracy: 0.8867 - val_loss: 0.3772
Epoch 133/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.1131
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1731 - val_accuracy: 0.8667 - val_loss: 0.4757
Epoch 134/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.3830
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9167 - loss: 0.2392 - val_accuracy: 0.8733 - val_loss: 0.4411
Epoch 135/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1800
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1679 - val_accuracy: 0.8733 - val_loss: 0.3619
Epoch 136/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1266
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1733 - val_accuracy: 0.8733 - val_loss: 0.3609
Epoch 137/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1869
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1788 - val_accuracy: 0.8800 - val_loss: 0.3694
Epoch 138/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2098
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1487 - val_accuracy: 0.8667 - val_loss: 0.3834
Epoch 139/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.2497
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2418 - val_accuracy: 0.8467 - val_loss: 0.3855
Epoch 140/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.1800
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1436 - val_accuracy: 0.8733 - val_loss: 0.4298
Epoch 141/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.1188
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1301 - val_accuracy: 0.8867 - val_loss: 0.3695
Epoch 142/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1014
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1528 - val_accuracy: 0.9133 - val_loss: 0.3866
Epoch 143/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.3194
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1713 - val_accuracy: 0.8733 - val_loss: 0.3945
Epoch 144/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.2353
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1374 - val_accuracy: 0.8733 - val_loss: 0.3791
Epoch 145/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.0824
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1362 - val_accuracy: 0.8733 - val_loss: 0.3521
Epoch 146/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1502
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1302 - val_accuracy: 0.8533 - val_loss: 0.3980
Epoch 147/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.2224
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1439 - val_accuracy: 0.8800 - val_loss: 0.3544
Epoch 148/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1477
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1478 - val_accuracy: 0.9000 - val_loss: 0.3429
Epoch 149/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0758
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1257 - val_accuracy: 0.8867 - val_loss: 0.3701
Epoch 150/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1144
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1357 - val_accuracy: 0.8600 - val_loss: 0.4255
Epoch 151/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.0601
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9200 - loss: 0.2029 - val_accuracy: 0.8400 - val_loss: 0.6045
Epoch 152/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8750 - loss: 0.3110
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9183 - loss: 0.2374 - val_accuracy: 0.8667 - val_loss: 0.4546
Epoch 153/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1500
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9483 - loss: 0.1502 - val_accuracy: 0.8533 - val_loss: 0.4463
Epoch 154/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1372
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1924 - val_accuracy: 0.8600 - val_loss: 0.4931
Epoch 155/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.0900
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.1183 - val_accuracy: 0.8667 - val_loss: 0.3441
Epoch 156/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.1636
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9683 - loss: 0.1272 - val_accuracy: 0.8600 - val_loss: 0.4161
Epoch 157/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.0835
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1448 - val_accuracy: 0.9133 - val_loss: 0.3316
Epoch 158/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1127
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1349 - val_accuracy: 0.9000 - val_loss: 0.3447
Epoch 159/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.1606
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1386 - val_accuracy: 0.8467 - val_loss: 0.4262
Epoch 160/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0607
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1285 - val_accuracy: 0.8733 - val_loss: 0.4013
Epoch 161/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.2281
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1569 - val_accuracy: 0.8933 - val_loss: 0.3433
Epoch 162/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0955
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9633 - loss: 0.1140 - val_accuracy: 0.9133 - val_loss: 0.3470
Epoch 163/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0824
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1115 - val_accuracy: 0.8733 - val_loss: 0.4099
Epoch 164/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1225
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1119 - val_accuracy: 0.8800 - val_loss: 0.3966
Epoch 165/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2757
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1253 - val_accuracy: 0.8933 - val_loss: 0.3710
Epoch 166/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0508
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1220 - val_accuracy: 0.9200 - val_loss: 0.3634
Epoch 167/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0946
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1440 - val_accuracy: 0.8267 - val_loss: 0.4644
Epoch 168/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0493
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1535 - val_accuracy: 0.8800 - val_loss: 0.3951
Epoch 169/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0833
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1426 - val_accuracy: 0.8600 - val_loss: 0.4362
Epoch 170/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1463
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1162 - val_accuracy: 0.8867 - val_loss: 0.4485
Epoch 171/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2356
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1530 - val_accuracy: 0.8467 - val_loss: 0.5432
Epoch 172/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0824
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.1909 - val_accuracy: 0.8533 - val_loss: 0.5197
Epoch 173/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0408
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.2217 - val_accuracy: 0.8733 - val_loss: 0.4418
Epoch 174/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0748
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1708 - val_accuracy: 0.8667 - val_loss: 0.4588
Epoch 175/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.1476
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.1943 - val_accuracy: 0.8600 - val_loss: 0.4101
Epoch 176/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2410
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1579 - val_accuracy: 0.8733 - val_loss: 0.4058
Epoch 177/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.2210
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1148 - val_accuracy: 0.8933 - val_loss: 0.3906
Epoch 178/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.0869
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1298 - val_accuracy: 0.8800 - val_loss: 0.3728
Epoch 179/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.0777
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1702 - val_accuracy: 0.8600 - val_loss: 0.4672
Epoch 180/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0833
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1723 - val_accuracy: 0.8600 - val_loss: 0.3896
Epoch 181/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.2012
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1583 - val_accuracy: 0.8800 - val_loss: 0.3280
Epoch 182/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1399
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1415 - val_accuracy: 0.9000 - val_loss: 0.2964
Epoch 183/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0563
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9683 - loss: 0.1090 - val_accuracy: 0.8867 - val_loss: 0.4197
Epoch 184/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.2035
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1254 - val_accuracy: 0.9067 - val_loss: 0.3570
Epoch 185/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1098
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9717 - loss: 0.0943 - val_accuracy: 0.9000 - val_loss: 0.3463
Epoch 186/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0878
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9783 - loss: 0.0912 - val_accuracy: 0.9067 - val_loss: 0.3375
Epoch 187/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0596
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1092 - val_accuracy: 0.8933 - val_loss: 0.3668
Epoch 188/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0886
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1257 - val_accuracy: 0.9000 - val_loss: 0.3677
Epoch 189/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.0840
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1255 - val_accuracy: 0.9000 - val_loss: 0.3372
Epoch 190/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0894
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9733 - loss: 0.1025 - val_accuracy: 0.8933 - val_loss: 0.3640
Epoch 191/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0625
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.0969 - val_accuracy: 0.9067 - val_loss: 0.3596
Epoch 192/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0570
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9633 - loss: 0.0953 - val_accuracy: 0.8933 - val_loss: 0.3582
Epoch 193/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.0680
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.0954 - val_accuracy: 0.9133 - val_loss: 0.3517
Epoch 194/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.1028
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9750 - loss: 0.0914 - val_accuracy: 0.8933 - val_loss: 0.3698
Epoch 195/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0420
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9767 - loss: 0.0953 - val_accuracy: 0.8933 - val_loss: 0.3885
Epoch 196/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0618
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.0904 - val_accuracy: 0.9067 - val_loss: 0.3603
Epoch 197/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.0930
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9767 - loss: 0.0885 - val_accuracy: 0.8667 - val_loss: 0.3615
Epoch 198/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.0303
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1208 - val_accuracy: 0.9067 - val_loss: 0.3669
Epoch 199/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.1255
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1328 - val_accuracy: 0.8733 - val_loss: 0.3895
Epoch 200/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.1078
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1425 - val_accuracy: 0.8800 - val_loss: 0.4179
```

In diesem Beispiel ist `X` die Designmatrix, `y` der Ergebnisvektor, und das Argument `epochs` gibt an, wie viele Epochen der Optimierer durchlaufen soll. *Keras* gibt den Fortschritt beim Optimieren an, es könnte so aussehen

```
Epoch 1/200
25/25 [==============================] - 0s 2ms/step - loss: 1.5984 - accuracy: 0.2438
Epoch 2/200
25/25 [==============================] - 0s 2ms/step - loss: 1.5709 - accuracy: 0.2763
Epoch 3/200
25/25 [==============================] - 0s 3ms/step - loss: 1.5550 - accuracy: 0.3013
```

Hierbei kann man die aktuelle Epoche, den Batch (für stochastischen Gradientenabstieg) und die aktuelle Genauigkeit der Trainingsdaten sehen. In diesem Beispiel werden alle Epochen bis zum Abschluss ausgeführt, daher sehen wir `32/32` für Batches, aber normalerweise kann man die Batch-Nummer sehen, die sich während des Trainings fortschreitet. Dieses Beispiel funktioniert sehr schnell, Keras meldet 2 ms pro Schritt (Batch) und die Gesamtzeit für die Epoche ist zu gering, um gemeldet zu werden. Aber eine einzelne Epoche kann bei komplexeren Modellen und mehr Daten viele Minuten dauern.

⚠ Achtung: Batchqualität vs. Modellqualität

Die während des Trainings angezeigte Modellqualität (hier Genauigkeit) ist eine laufende Trainingsmetrik und kann zwischen Batches und Epochen schwanken. Sie gibt nur eine grobe Richtlinie für die tatsächliche Modellgüte und sollte durch eine getrennte Auswertung auf Validierungs- oder Testdaten ergänzt werden.

⚠ Achtung: Modelle fertig trainieren

Keras ermöglicht es Ihnen, Vorhersagen mit einem Modell zu treffen, das nicht angepasst ist (im Gegensatz zu scikit-learn, wo das einen Fehler verursacht). Die Ergebnisse werden bestenfalls mittelmäßig aussehen.

Vorhersagen und Plotten

Wenn die Anpassung abgeschlossen ist, können wir das Modell für **Vorhersagen** verwenden. Die Vorhersage funktioniert ähnlich wie in sklearn, nur dass die `predict` -Methode die Wahrscheinlichkeit vorhersagt, nicht die Kategorie (analog zu sklearn's `predict_proba`).

```
phat = model.predict(X_test)
```

```
1/8 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step

8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```

In diesem Beispiel handelt es sich um eine Matrix mit 5 Spalten, wobei jede Spalte die Wahrscheinlichkeit darstellt, dass der Datenpunkt zur entsprechenden Kategorie gehört. Beispielsweise könnten Zeilen von `phat` so aussehen:

```
phat[:5]
```

```
array([[1.3095678e-30, 1.1563733e-11, 6.0592359e-01, 3.9158618e-01,
        2.4902199e-03],
       [7.4105847e-01, 2.5893581e-01, 5.7627967e-06, 5.3906651e-15,
        4.6460556e-28],
       [9.9994993e-01, 5.0025272e-05, 3.8763662e-11, 1.3828770e-20,
        1.2327348e-31],
       [9.8848331e-01, 1.1516372e-02, 1.9062206e-07, 3.0599248e-16,
        1.5762319e-24],
       [9.9999130e-01, 8.6841928e-06, 6.6179341e-14, 3.3155176e-25,
        0.0000000e+00]], dtype=float32)
```

Wir sehen, dass für jedes Test-Fall eine Wahrscheinlichkeit für jede Kategorie angegeben ist. Als nächstes wollen wir mit `np.argmax(phat, axis=-1)` die Spalte (Kategorie) finden, mit der höchsten Wahrscheinlichkeit. Es findet einfach die Position der größten Elemente im Array entlang der letzten Achse (`axis=-1`), d.h. Spalten. Für jede Zeile finden wir also die entsprechende Spaltennummer. Beachten Sie, dass `np.argmax` die Spalten ab 0 zählt, nicht ab 1:

```
yhat = np.argmax(phat, axis=-1)
yhat[:5]
```

```
array([2, 0, 0, 0, 0])
```

Endlich können wir die Verwirrungsmatrix mit `pd.crosstab` berechnen und die Genauigkeit berechnen:

```
from sklearn.metrics import confusion_matrix

print("confusion matrix:\n", confusion_matrix(y_test, yhat))
print("Accuracy (on test data):", np.mean(y_test == yhat))
```

```
confusion matrix:
 [[51  2  1  0  0]
 [ 5 23  5  0  1]
 [ 1  2 60  7  0]
 [ 0  0  2 31  5]
 [ 0  0  0  1 53]]
Accuracy (on test data): 0.872
```

In diesem Beispiel machen wir Vorhersagen auf Testdaten, aber wir können natürlich auch einen anderen Datensatz für Vorhersagen auswählen. Da der vorhergesagte Wert eine Wahrscheinlichkeitsmatrix mit 5 Spalten sein wird, berechnen wir `yhat` als die Spaltennummer, die die größte Wahrscheinlichkeit für jede Zeile enthält.

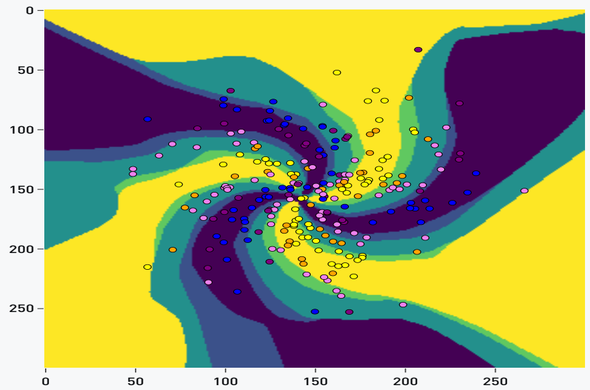
Wie man sehen kann, ist die Verwirrungsmatrix fast ausschließlich auf der Hauptdiagonalen besetzt, und die Genauigkeit ist hoch.

Keras-Vorhersagen

Schließlich müssen wir, wenn wir ein ähnliches Diagramm wie oben erstellen möchten, die `DBPlot` -Funktion anpassen, um zu berücksichtigen, dass Keras-Modelle nur Wahrscheinlichkeiten vorhersagen.

```
def DBPlot(m, X, y, nGrid = 300):
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, nGrid),
                           np.linspace(x2_min, x2_max, nGrid))
    XX = np.column_stack((xx1.ravel(), xx2.ravel()))
    ## predict probability
    phat = m.predict(XX, verbose=0)
    ## find the column that corresponds to the maximum probability
    hatyy = np.argmax(phat, axis=-1).reshape(xx1.shape)
    fig = px.imshow(hatyy, width=600, height=600)
    fig.add_scatter(x=X[:,0]/(x1_max-x1_min)*nGrid+nGrid/2, y=X[:,1]/(x2_max-x2_min)*nGrid+nGrid/2, mode="markers"
                    marker=dict(color=[mcolors[i] for i in y]), marker_line=dict(width=.3, color="black"))
    fig.update_coloraxes(showscale=False)
    fig.update_layout(showlegend=False)
    fig.show()
```

```
DBPlot(model, X_test, y_test)
```



Noch einmal, die Funktion ist fast identisch mit der *sklearn* Version, außer der Zeile, die `np.argmax(phat, axis=-1)` enthält, die die vorhergesagten Wahrscheinlichkeiten in Kategorien umwandelt.

Referenzen

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115–133.
Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), 386.

QUESTIONS